VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA LADICÍCH INFORMACÍ V SESTAVUJÍCÍM PROGRAMU

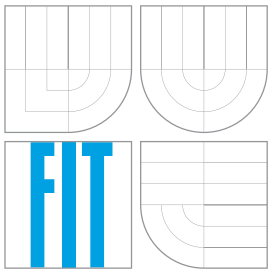
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH NIKL

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA LADICÍCH INFORMACÍ V SESTAVUJÍCÍM PROGRAMU

DEBUGGING INFORMATION IN LINKER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH NIKL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2012

Abstrakt

Tato práce popisuje převod objektového formátu CCOFF do formátu ELF a zpět. Nejdříve je popsán obecný formát objektového souboru a využité ladicí informace DWARF, poté konkrétněji formát CCOFF a ELF. Veškerá funkcionalita spojená s manipulací s formátem CCOFF je zapouzdřena v kolekci tříd ObjectFile. V práci je popsán způsob vytvoření ELF objektového souboru a jeho naplnění korektními daty a následně zpětná konverze zpět do formátu CCOFF.

Abstract

This thesis describes the conversion between the CCOFF object file format and the ELF file format. We start with a general object file format and its debbuging information, then we focus closely on the ELF, CCOFF and DWARF debugging information. The functionality of the CCOFF format is encapsulated in the ObjectFile class library. Then follows the description of creating an ELF object file, its filling with the proper data and its conversion back to the CCOFF format.

Klíčová slova

Lissom, Codasip, ELF, CCOFF, konvertor, symbol, sekce, objektový soubor, relokace, řádkové informace, tabulka symbolů, sestavující program, DWARF, DIE

Keywords

Lissom, Codasip, ELF, CCOFF, converter, symbol, section, object file , relocation, line number information, symbol table, linker, DWARF, DIE

Citace

Vojtěch Nikl: Podpora ladicích informací v sestavujícím programu, bakalářská práce, Brno, FIT VUT v Brně, 2012

Podpora ladicích informací v sestavujícím programu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Karla Masaříka, Ph.D. Další cenné informace mi poskytli Ing. Adam Husár a Bc. Peter Matula. Uvedl jsem všechny literální prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Nikl
14. května 2012

Poděkování

Na tomto místě bych rád poděkoval Ing. Karlu Masaříkovi, Ph.D. za vedení bakalářské práce, Ing. Adamovi Husárovi za trpělivost, věnovaný čas a velké množství cenných rad. Dále děkuji Bc. Peteru Matulovi za poskytnutí dalších užitečných informací a odkazů.

Obsah

Kapitola 1

Úvod

Málokterý program dnes vzniká přeložením jediného zdrojového souboru. I ty nejmenší projekty se skládají z menšího či většího množství modulů, které je potřeba samostatně přeložit a poté spojit do výsledného programu. O spojení se stará spojující program (angl. linker), který je součástí většiny dnešních překladačů. Jelikož se ale nejedná o žádnou triviální úlohu, musí vstupní soubory splňovat přesně specifikovaný formát. Princip linkerů je ale ve všech případech stejný, vezmou obsah vstupních souborů, provedou potřebné modifikace a uloží výsledek do jednoho výstupního (nejčastěji přímo spustitelného) souboru.

Formátů pro popis objektových souborů existuje celá řada. Jedním z nich je i formát CCOFF (Common Codasip Object File Format) využívaný v projektu Lissom. Mít vlastní specifický formát ale zároveň znamená mít i vlastní linker. Vývoj a ladění takového nástroje je časově i finančně velmi náročný, navíc aktuální linker nemá podporu pro ladicí informace, a proto se zvažovaly dvě možnosti. Rožšíření stávajícího linkeru nebo využití GNU linkeru, který již podporu pro ladicí informace obsahuje. S využití GNU linkeru by se ale zároveň musel překonvertovat formát CCOFF do formátu ELF.

Po konzultaci s vedoucím byla vybrána druhá možnost. GNU linker již existuje delší dobu a jedná se o plně funkční a odladěný nástroj. Formát ELF poskytuje vysokou rychlost při zpracování, překladu a linkování, nativní podporu dynamického linkování knihoven a jazyka C++, tzv. weak symboly, které řeší konflikt stejně pojmenovaných symbolů ve více různých přilinkovaných knihovnách, binární podobu souboru rozděleného do bloků (sekcí a segmentů) atd. Zároveň jsou zde i některé nevýhody, např. menší univerzálnost, složitější úpravy a modifikace funkčnosti a nutnost vytvořit konvertor z formátu CCOFF do formátu ELF.

Tím pádem je, se souhlasem vedoucího, cílem této bakalářské práce navrhnout úpravy formátu ELF tak, aby byly uchovány všechny informace z formátu CCOFF, implementovat konvertor pro převod z CCOFFu do tohoto formátu a nazpět a tím zajistit podporu ladicích informací.

Kapitola 2

Objektový soubor

Programy napsané ve vyšších programovacích jazycích (C, C++, ...) jsou čitelné pro člověka, nikoli ale pro samotný stroj, který tyto programy vykonává. Proto musí proběhnout několik fází, než je možné program spustit. Např. v případě jazyka C je prvním krokem zpracování preprocesorem, který se stará o nahrazení textu, odstranění komentářů a inkluzi jiných souborů. Druhým krokem je kompilace zdrojového kódu do jazyka symbolických instrukcí, v tomto případě assembleru. Po tomto kroce je program stále čitelný pro člověka, poskytuje ale velmi nízkou úroveň abstrakce. Třetím krokem je vytvoření objektového souboru. Ten již obsahuje strojový kód, který může zpracovat procesor, a další informace potřebné pro sestavení více objektových souborů pomocí linkeru do jednoho, nejčastěji spustitelného, souboru. V následující části se zaměřím na to, jak takový objektový soubor vypadá.

2.1 Objektový soubor

Obecný formát objektového souboru obvykle obsahuje těchto 5 informací.

- **Hlavička souboru** specifikuje základní informace o souboru a sémantiku dat.
- **Objektový kód** obsahuje binárná instrukce a data, které vygeneroval překladač nebo assembler.
- **Relokace** neboli seznam míst v objektovém souboru, které obsahují nevyřešené odkazy a v době sestavování musí být nahrazeny konkrétními adresami.
- **Symboly** mohou být globální definované v konkrétním modulu, importované z jiného modulu nebo definované linkerem.
- **Ladicí informace** jsou potřebné pro debugger při ladění programu. Patří sem informace o řádcích, lokální symboly, deklarace speciálních typů atd.

2.2 Typ objektového souboru

Objektový soubor může být *relokatibilní*, tedy obsahovat informace relokační informace, o které se musí postarat linker. Dále může být *spustitelný*, lze ho tedy nahrát do paměti a spustit. Nebo jako forma *sdílené knihovny*, která se nahraje do paměti společně s hlavním programem. Může jít i o kombinaci těchto tří typů. Některé formáty objektových souborů podporují pouze jeden nebo dva typy, některé všechny tři.

2.3 Formáty objektových souborů

V další části se podíváme na nejznámější a nejpoužívanější formáty, ze kterých vycházejí dnes používané formáty, které byly použity při řešení této práce. Více informací lze nalézt v [1].

2.3.1 MS-DOS .COM

Pravděpodobně nejjednodušší formát pro architekturu x86, jaký byl v minulosti využíván. .COM obsahuje pouze spustitelný binární kód. Když operační systém spustí soubor .COM, celý program se načte do volné paměti na offset 0x100. Všechny segmentové registry procesoru se nastaví tak, aby ukazovaly do jediného segmentu PSP (Program Segment Prefix), registr SP (Stack Pointer) pak ukazuje na konec segmentu, protože zásobník roste směrem k nižším adresám. Pokud se program nevešel do jediného segmentu, byla práce programátora poradit si s bitovými opravami a umožnit tak programu běh na více segmentech.

2.3.2 a.out

Trochu složitější, ale dříve velmi využívaný, je unixový formát a.out. V nejjednodušším případě se program skládal z malé hlavičky, spustitelného binárního kódu (z historických důvodů uloženém v sekci s názvem *text*) a výchozích hodnot statických proměnných. Využívalo se 16bitové adresování, díky čemuž byl program limitován velikostí 64kB. Tento limit byl velice brzy příliš omezující, a proto se adresní prostor rozdělil na prostor pro kód a pro data. Překladače, assemblery a linkery musely být upraveny tak, aby zvládaly práci se dvěma různými sekcemi ve dvou různých paměťových segmentech (později přibyly i další sekce jako tabulka symbolů, tabulka řetězců atd.). Tento přístup byl i efektivnější díky tomu, že kopie toho samého programu mohly sdílet jednu kopii programového kódu, který se neměnil, a ušetřilo se díky tomu nemalé množství paměti. Poslední systém, který využíval takto oddělené adresování pro kód a pro data, byl 80286 a 80386 v 16bitovém chráněném režimu.

Ne všechny systémy ale načítají programy na stejné logické adrese, a proto se do programu musely přidat relokační informace (často nazývané anglicky jako *fix-ups*), které označují, jaká místa v paměti musejí být modifikována při sestavování.

2.3.3 ELF

Klasický a.out formát sloužil unixové komunitě více než 10 let, ale s příchodem systému UNIX V se společnost AT&T rozhodla, že pro lepší podporu přenositelnosti, dynamického linkování a dalších novinek, potřebují nový formát.Nejdříve se začal používat formát COFF (Common Object File Format), ten ale nativně nepodporoval C++ a dynamické linkování, proto se přešlo k formátu ELF (Executable and Linkable Format). ELF je standardní souborový formát pro popis relokatibilních a spustitelných objektových souborů, sdílených knihoven a ladicích výpisů. Byl původně vyvinut v Unix System Laboratories pro operační systém UNIX System V [2]. V současné době je poměrně rozšířený a využívají ho operační systémy jako Linux, FreeBSD, OpenBSD, Solaris, Symbian a další. Podrobněji se na něj zaměříme v kapitole [4]. Je s ním úzce spjatý mj. formát DWARF (Debug With Arbitrary Record Format) pro uložení ladicích informací.

Kapitola 3

Ladicí informace DWARF

Při ladění programu využíváme některé velmi užitečné funkce a operace, jako např. zarážky, krokování po řádcích nebo po blocích, zobrazování obsahu proměnných atd. Aby bylo možné tyto možnosti využívat, musí být v programu uloženy dodatečné ladicí informace. Jedním z nejpoužívanějších formátů je DWARF [3]. Kompletní dokumentaci lze nalézt v [4].

Většina dnešních moderních programovacích jazyků je strukturována do bloků. Každá entita (např. definice třídy nebo funkce) se nachází uvnitř jiné entity. Překladače proto velmi často vnitřně reprezentují program jako strom.

DWARF má velmi podobnou strukturu. Každa entita (kromě těch na nejvyšší úrovni obsahující informace o souboru) je obsažena uvnitř své rodičovské entity a může mít své potomky a sourozence. Vnitřní popis programu je tedy také reprezentován jako strom. Díky tomu může být DWARF využíván u téměř libovolného procedurálního jazyka na jakékoli architektuře, nejčastěji se ale využívá společně s objektovým formátem ELF.

Základní jednotkou pro popis DWARFu je tzv. DIE (Debugging Information Entry). Ta obsahuje svůj *tag* (označení), který specifikuje typ jednotky, a seznam atributů, které blíže popisují její obsah. Obrázek 3.1 graficky znázorňuje, jakým způsobem je popsán jednoduchý program na 32bitové architektuře. DIE nejvyšší úrovně popisuje *kompilační jednotku*, jeho dva potomci popisují hlavní funkci a typ celého čísla.

DIE se dělí na dva hlavní typy. První popisující data včetně datových typů a druhý popisující funkce a další spustitelný kód.

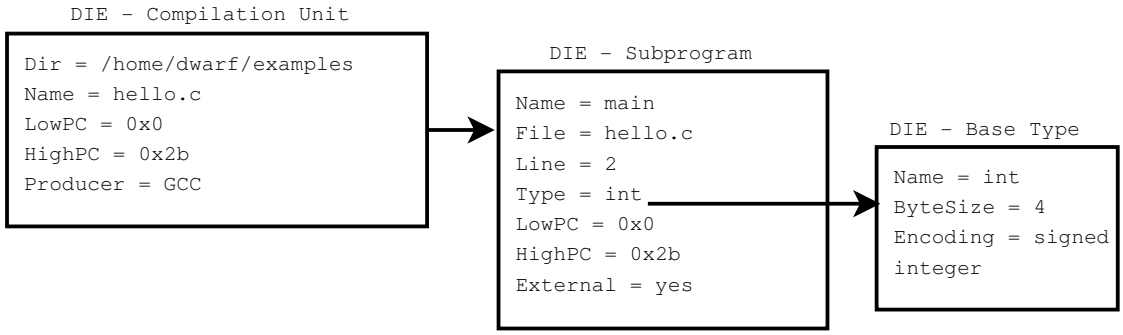
3.1 DIE popisující data a jejich typy

Mnoho programovacích jazyků má vlastní sofistikovaný popis dat. Obsahují množství zabudovaných typů, ukazatelů, datových struktur a možnost vytvořit si vlastní uživatelské typy. Aby byl DWARF se všemi těmito možnostmi kompatibilní, obsahu *základní typy* určené přímo pro hardware a *složené typy* skládající se z kombinací základních typů.

3.1.1 Základní typy

Základní typy poskytují mapování mezi nejjednoduššími datovými typy, jako jsou např. celé nebo desetinné číslo, a způsobem, jakým jsou implementovány na cílovém hardwaru. Tabulka 3.1 ukazuje, jak lze uložit 16bitovou proměnnou na 32bitové architektuře. 16bitová hodnota se ulozí do spodních 16bitů 32bitového slova.

```
hello.c:
        1: int main()
        2: {
        3:   printf("Hello World!\n");
        4:   return 0;
        5: }
```



Obrázek 3.1: Grafická reprezentace DWARF dat. Převzato z [3].

```
DW_TAG_base_type
  DW_AT_name = word
  DW_AT_byte_size = 4
  DW_AT_bit_size = 16
  DW_AT_bit_offset = 0
  DW_AT_encoding = signed
```

Tabulka 3.1: Deklarace 16bitové proměnné na 32bitové architektuře v DWARFu. Převzato z [3].

3.1.2 Složené typy

Pole Pole je popsáno pomocí DIE, které definuje, zda jsou data uložena po řádcích nebo po sloupcích. Index pole je reprezentován jako rozsah, který udává spodní a horní hranici každé dimenze pole. Díky tomu může DWARF popsat jak pole jazyka C, které začínají vždy indexem nula, tak i pole použité např. v Pascalu, kde spodní hranice může začínat na libovolném indexu.

Struktury, třídy, uniony a rozhraní Většina jazyků umožňuje programátorovi seskupit data do struktur, např. *struktura* v jazyce C, *třída* v C++ nebo *záznam* v Pascalu. *Union* je speciální typ struktury, kde všechny prvky sdílejí jeden adresový prostor a velikost unionu je dána velikostí největšího prvku. Rozhraní, používané v Javě je podmnožinou tříd jazyka C++, protože může obsahovat pouze abstraktní metody a konstantní datové členy.

DWARF má velmi podobnou terminoologii, jako jazyk C++, a definuje DIE popisující strukturu, union, třídu a rozhraní. V další části se za-

měříme na DIE popisující třídu, ostatní mají ale velmi podobnou organizaci.

DIE popisující třídu je rodičem všech DIE popisující členy třídy. Každá třída má své jméno a pravděpodobně i další atributy. Pokud je velikost dané třídy známa již v době překladu, obsahuje také atribut *bytová velikost*. Popis každého atributu má velmi podobný tvar jako popis proměnné, ačkoli zde mohou být některé další atributy jako kvalifikátory public, private a protected v C++, které vyžadují speciální popis.

Proměnné

Proměnnou můžeme chápat jako pojmenovaný kus paměti nebo registr. Rozsah hodnot a možné operace jsou definovány typem proměnné. Co proměnné skutečně rozlišuje, je umístění proměnné v programu a rozsah viditelnosti. Díky tomu může být více proměnných pojmenováno stejně a nezpůsobit přitom konflikt při překladu. DWARF ukládá informace o jejím umístění jako trojici soubor, řádek a sloupec.

DWARF rozděluje proměnné do tří kategorií: konstanty, parametry funkce a ostatní proměnné. V některých jazycích mohou být proměnné deklarovány, ale nemusí být definovány. Proto i DIE popisující proměnnou nemusí obsahovat umístění její definice. Většina proměnných obsahuje informace o umístění proměnné, v nejjednodušším případě je umístěna v paměti a má pevnou adresu, resp. pevný offset od začátku souboru nebo sekce. Lokální proměnné jsou však alokovány dynamicky za běhu a mohou být umístěny na zásobníku nebo v registru, což vyžaduje složitější výpočty pro určení finálního umístění.

DWARF ukládá údaje o umístění proměnných jako sekvenci operací, pomocí nichž dokáže debugger lokalizovat data. Na obrázku 3.2 je ukázáno, jak tyto údaje vypadají. Proměnná `a` má pevnou adresu, proměnná `b` je registr 0 a proměnná `c` se nachází na offsetu -12 aktuálního zásobníku funkce. Přestože `a` bylo deklarováno jako první, DIE popisující `a` bude generován až později. Skutečná adresa bude poté doplněna linkerem.

3.2 Spustitelný kód

DWARF popisuje funkce a procedury velice podobně pomocí DIE s názvem *Subprogram*, který obsahuje jméno, lokaci, informaci, zda se jedná o externí symbol, lokace obsažených dat, DIE s typy parametrů, adresu, kam se vrátit po skončení funkce atd. Funkce může být uložena buď v souvislém bloku paměti nebo na více paměťových místech. DWARF nicméně nepopisuje konveci volání funkce, ta je definována v ABI (Application Binary Interface).

3.3 Práce s více soubory

Téměř všechny programy se skládájí z více zdrojových souborů, které jsou překládány individuálně a o spojení do výsledného programu se postará linker. Podobně i DWARF má pro každý soubor DIE typu *Compilation unit* kompilační jednotka. Tato jednotka obsahuje mj. jméno zdrojového souboru i s cestou, použitý programovací jazyk a offsety jednotlivých datových sekcí s DWARF informacemi 3.1.

```
example.c:
        1: int a;
        2: void foo()
        3: {
        4:   register int b;
        5:   int c;
        6: }

<1>: DW_TAG_subprogram
       DW_AT_name = foo

<2>: DW_TAG_variable
       DW_AT_name = b
       DW_AT_type = <4>
     DW_AT_location = (DW_OP_reg0)

<3>: DW_TAG_variable
       DW_AT_name = c
       DW_AT_type = <4>
       DW_AT_location = (DW_OP_fbreg: -12)

<4>: DW_TAG_base_type
       DW_AT_name = int
       DW_AT_byte_size = 4
       DW_AT_encoding = signed

<5>: DW_TAG_variable
       DW_AT_name = a
       DW_AT_type = <4>
       DW_AT_external = 1
       DW_AT_location = (DW_OP_addr: 0)
```

Obrázek 3.2: Reprezentace proměnných v DWARFu. Převzato z [3].

3.4 Zakódování dat

DWARF informace ve své první verzi zabíraly velké množství místa. Ve druhé verzi už se dokázal prostor ušetřit díky tomu, že strom reprezentující program byl uložen v prefixovém pořadí.

Každý typ DIE má předem určeno, zda může mít potomky. Pokud ano, další DIE je jeho potomkem, pokud ne, je jeho sourozencem. Díky tomu mohou být odkazy na sourozence, resp. potomky zcela eliminovány. Pokud se překladač potřebuje dostat k sourozencům aktuálního DIE bez toho, aby musel procházet všechny jeho potomky, může se v DIE uchovávat atribut s odkazem na sourozence.

Další možností, jak uspořit cenné místo, je použití zkratek. DWARF umožnuje generovat DIE i s jejich atributy automaticky, vetšina překladačů ale dokáže generovat pouze omezenou sadu DIE se stále se opakující sadou atributů. Místo ukládání dvojice atribut–hodnota pokaždé zvlášť je uložen pouze index do tabulky zkratek následovaný kódem atributu.

Méně využívanou možností, uvedenou ve třetí verzi, jsou přímé odkazy z jedné kompilační jednotky do dat druhé kompilační jednotky nebo sdílené knihovny. Většina překladačů generuje pokaždé stejnou tabulku zkratek bez ohledu na to, zda jsou všechny zkratky skutečně využívány. Stačí tedy mít jednu tabulku zkratek uloženou např. ve sdílené knihovny a ostatní kompilační jednotky na ni odkázat.

3.5 Řádkové informace

Obsahují mj. informace o čísle řádku a odpovídající adrese první instrukce na daném řádku. Tyto údaje jsou uloženy v tabulce, kde první sloupec obsahuje adresy a další pak trojici soubor, řádek, sloupec. Pokud chceme do programu vložit zarážku, tabulka nám vrátí odpovídající adresu instrukce. Ukázka takové tabulky je uveden v tabulce 3.2.

Address	File	Line	Col	Stmt	Block	End	Prolog	Epilog	ISA
0x0	0	42	0	yes	no	no	no	no	0
0x9	0	44	0	yes	no	no	no	no	0
0x1a	0	44	0	yes	no	no	no	no	0
0x24	0	46	0	yes	no	no	no	no	0
0x2c	0	47	0	yes	no	no	no	no	0
0x32	0	49	0	yes	no	no	no	no	0
0x41	0	50	0	yes	no	no	no	no	0
0x47	0	51	0	yes	no	no	no	no	0
0x50	0	53	0	yes	no	no	no	no	0
0x59	0	54	0	yes	no	no	no	no	0
0x6a	0	54	0	yes	no	no	no	no	0
0x73	0	55	0	yes	no	no	no	no	0
0x7b	0	56	0	yes	no	yes	no	no	0

File 0: `strndup.c`

Tabulka 3.2: Příklad řádkových informací. Převzato z [3].

Pokud by každá instrukce měla mít svůj vlastní řádek, velikost takové tabulky by mnohonásobně převýšila velikost samotného programu. DWARF proto tabulku reprezentuje jako posloupnost instrukcí nazývaných *line number program*, které zamezují ukládání opakujících se informací z tabulky. Tyto instrukce jsou interpretovány jednoduchým konečným automatem pro znovu vytvoření původních záznamů.

Kapitola 4

Formát ELF

Existují 3 základní typy ELF souborů.

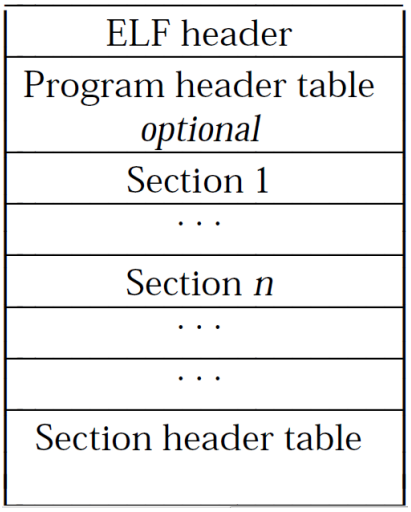
relokatibilní Obsahuje nevyřešené odkazy v paměti a před spuštěním musí být zpracován sestavovacím programem. Tímto typem se budeme zabývat v dalších částech práce nejvíce.

spustitelný Má vyřešeny všechny relokace a odkazy na symboly, kromě symbolů ze sdílených knihoven, které jsou vyřešeny v době běhu programu.

sdílená knihovna Obsahuje symboly a binární kód, spojuje se staticky nebo dynamicky s hlavním programem.

Podrobnější informace lze nalézt v manuálových stránkách [5] nebo v dokumentu detailně popisujícím jednotlivé položky formátu a práci se souborem [6].

První část ELF souboru tvoří hlavička souboru, následuje volitelná programová hlavička, obsahy sekcí a tabulka hlaviček sekcí, jak je znázorněno na obrázku 4.1.



Obrázek 4.1: Grafické zobrazení obecného ELF souboru. Převzato z [6].

4.1 Hlavička souboru

Nalézá se na začátku souboru a její typ je deklarován následovně.

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half e_type;
  Elf32_Half e_machine;
  Elf32_Word e_version;
  Elf32_Addr e_entry;
  Elf32_Off  e_phoff;
  Elf32_Off  e_shoff;
  Elf32_Word e_flags;
  Elf32_Half e_ehsize;
  Elf32_Half e_phentsize;
  Elf32_Half e_phnum;
  Elf32_Half e_shentsize;
  Elf32_Half e_shnum;
  Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

`e_ident`	specifikuje základní informace o typu souboru, strojově nezávislá data a umístění dalších částí souboru,
`e_type`	typ objektového souboru,
`e_machine`	cílovou architekturu,
`e_version`	verzi objektového souboru,
`e_entry`	adresu vstupního bodu,
`e_phoff`	relativní adresu hlavičky programu vůči začátku souboru,
`e_shoff`	relativní adresu tabulky hlaviček sekcí vůči začátku souboru,
`e_flags`	specifické značky pro použitý procesor související s objektovým souborem,
`e_ehsize`	velikost ELF hlavičky v bytech,
`e_phentsize`	velikost záznamu v programové hlavičce v bytech,
`e_phnum`	počet záznamů v programové hlavičce,
`e_shentsize`	velikost hlavičky sekce v bytech,
`e_shnum`	počet záznamů v tabulce hlaviček sekcí,
`e_shstrndx`	index hlavičky sekce s textovými řetězci v tabulce hlaviček sekcí.

4.2 Hlavička sekce

Definuje jméno, vlastnosti sekce a dat v ní uložených. První sekce v souboru je nulová sekce, která má vsechny atributy vynulováné. Hlavička je deklarována jako následující struktura.

```
typedef struct
{
  Elf32_Word  sh_name;
  Elf32_Word  sh_type;
  Elf32_Word  sh_flags;
```

```
  Elf32_Addr  sh_addr;
  Elf32_Off   sh_offset;
  Elf32_Word  sh_size;
  Elf32_Word  sh_link;
  Elf32_Word  sh_info;
  Elf32_Word  sh_addralign;
  Elf32_Word  sh_entsize;
} Elf32_Shdr;
```

`sh_name`	obsahuje index do sekce s textovými řetězci se jménem sekce,
`sh_type`	typ sekce specifikující obsah a sémantiku,
`sh_flags`	příznaky popisující speciální atributy,
`sh_addr`	adresu prvního bytu sekce v případě uložení sekce do paměti,
`sh_offset`	relativní adresu od začátku souboru do prvního bytu sekce,
`sh_size`	velikost sekce i s jejími daty v bytech,
`sh_link`	index do tabulky hlaviček sekcí,
`sh_info`	informace, jejichž interpretace závisí na typu sekce, například sekce s tabulkou symbolů zde má uložený index sekce s textovými řetězci,
`sh_addralign`	zarovnání dat,
`sh_entsize`	velikost prvku sekce v bytech, pokud obsahuje pouze prvky s pevnou předem známou velikostí.

4.3 Tabulka symbolů

Jedná se o speciální typ sekce, která slouží k uložení všech identifikátorů ve zdrojovém kódu a jejich dodatečných informací, které slouží např. k identifikaci symbolu nebo uplatnění relokace. Všechny prvky mají pevně danou velikost `sizeof(Elf32_Sym)`. Nultý index je rezervován pro nultý prvek, který má všechny atributy nulové. Prvek tabulky symbolů má následující formát.

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

`st_name`	obsahuje index do tabulky jmen symbolů,
`st_value`	hodnotu symbolu, kde se může podle typu symbolu jednat o absolutní velikost, adresu v paměti atd.,
`st_size`	velikost symbolu v bytech,
`st_info`	specifikuje typ symbolu a jeho vazební atributy. Položka se vytvoří pomocí makra: `#define ELF32_ST_INFO(b,t) (((b) << 4) + ((t) & 0xf))` Zpětně se dá typ a vazba získat pomocí maker:

```
#define ELF32_ST_BIND(i)    ((i) >> 4)
#define ELF32_ST_TYPE(j)    ((j) & 0xf)
```

`st_other` prozatím nemá definovaný význam a obsahuje hodnotu 0,

`st_shndx` je index sekce, v souvislosti s níž je symbol definován.

4.4 Relokace

Relokace je proces propojení symbolických odkazů s jejich definicemi. Například pokud program zavolá funkci, příslušná instrukce musí předat řízení konkrétní adrese programu. Jinými slovy, spojovací program musí zajistit vyřešení nevyřešených odkazů a modifikováním sekcí dosadit správné adresy na správná místa. K tomu je potřeba, aby si sekce uchovávala data, pomocí nichž lze relokace provést. V ELFu existují dva typy pro uchování relokačních informací, `Elf32_Rel` a `Elf32_Rela`. Principem a typy relokací v ELFu se nebudeme dále zabývat, protože jsou závislé na konkrétní architektuře a tudíž jsou pro konvertor v této podobě nepoužitelné. Řešení tohoto problému je uvedeno v kapitole o implementaci.

```
typedef struct {
        Elf32_Addr  r_offset;
        Elf32_Word  r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr   r_offset;
        Elf32_Word   r_info;
        Elf32_Sword  r_addend;
} Elf32_Rela;
```

`r_offset` udává místo (adresu), na kterém se má provést příslušná relokace. V případě relokatibilních objektových souborů je adresa interpretována jako offset od začátku souboru, v případě spustitelných souborů nebo sdílených knihoven jde to virtuální adresu.

`r_info` uchovává dvě informace, index do tabulky symbolů spojený s konkrétní a typ prováděné relokace. Do jednoho slova se spojí pomocí makra:

```
#define ELF32_R_INFO(s,t) (((s) << 8) + (unsigned char)(t))
```

Zpětně se index a typ získá jako:

```
#define ELF32_R_SYM(i)     ((i) >> 8)
#define ELF32_R_TYPE(j)    ((unsigned char) (j))
```

`r_addend` je konstanta, která se připočítává u určitého typu relokace k relokované položce.

4.5 Tabulka textových řetězců

Nalézá se v sekci obvykle pojmenované jako *.strtab* nebo *.shstrtab* (tabulka jmen sekcí) a obsahuje všechny řetězce ukončené znakem `'\0'` použité v objektovém souboru. Linker využívá tyto řetězce pro reprezentaci jmen sekcí a symbolů.

První byte obsahuje ukončovací znak `'\0'`, řetězce jsou mezi sebou taktéž odděleny tímto znakem. Příklad obsahu této tabulky o velikosti 25 bytů je znázorněn v tabulce 4.2.

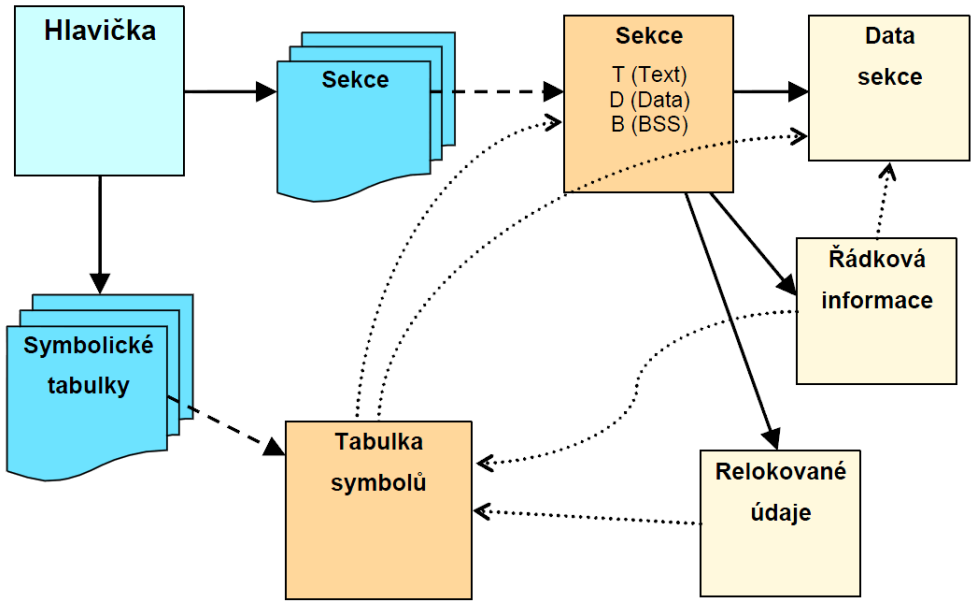
Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Obrázek 4.2: Ukázka obsahu tabulky textových řetězců. Převzato z [6].

Kapitola 5

Formát CCOFF

Výstupní formát pro assembler [7] a linker a zároveň vstupní formát pro linker, disassemler a simulátor, vycházející z formátu COFF, má název CCOFF (Common Codasip Object File Format). Jedná se o obecný popis souborů nezávislý na šířce slova instrukce, způsobu uložení čísel, charakteru instrukční sady a způsobu uložení instrukční sady a přístupu do ní. Prozatím se ukládá v textové formě, která je snadno čitelná pro člověka, na druhou stranu ale není příliš efektivní pro zpracování. Úplná specifikace formátu je uvedena v [8]. Struktura objektového souboru je graficky znázorněna na obrázku 5.1 .



Obrázek 5.1: Objektový soubor formátu CCOFF. Převzato z [8].

5.1 Obecné vlastnosti

Konec každého řádku je reprezentován znakem <LF> (0x0A). Všechny číselné údaje, týkající se orientace v souboru, jsou uloženy v dekadické hodnotě s požadovaným rozsahem 32 bitů, naopak číselné informace v kódových či datových sekcích jsou uloženy po řádcích v binárním kódu.

Textová informace je, z důvodu vyšší efektivity a zamezení postupné alokace zdrojů, uložena ve

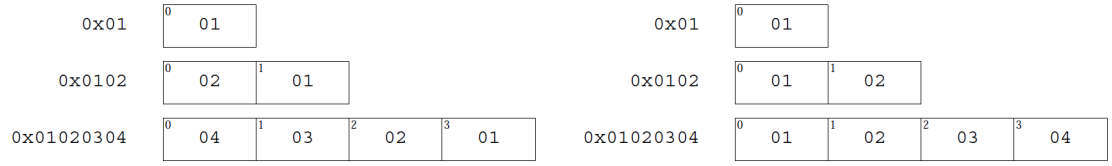
dvou řádcích, přičemž na prvním řádku se nachází velikost této informace bez ukončovacího řádku a na druhém je samotný text.

Celý formát je navržen tak, aby při zpracování nebylo nutné používat při načtení vícenásobného průchodu nebo postupné alokace zdrojů.

5.2 Hlavička souboru

Na začátku souboru se nachází povinná hlavička, která identifikuje CCOFF formát, architekturu a sémantiku uložených dat.

První údaj na prvním řádku je magický řetězec AgT62kG9y7, který sděluje, že jde o tento formát. Následují informace o architektuře. Nejmenší adresovatelnou jednotku určuje počet bitů ve slově podělený počtem bytů ve slově. Nejčastěji se setkáváme s 8bitovými byty, ale existují nebo existovaly i architektury se 7-bitovými nebo 14-bitovými byty. Způsob uložení dat určuje endianita, která má dva typy, little endian a big endian (obrázek 5.2).



Obrázek 5.2: Ukázka uložení čísla jako little (vlevo) a big (vpravo) endian. Převzato z [6].

Další důležité atributy jsou flags, které mj. sdělují, zda soubor obsahuje relativní informaci. Dále absolutní bytová adresa vstupního bodu, počet sekcí a počet tabulek symbolů.

5.3 Hlavička sekce

Hlavní jednotkou výstupního objektového souboru, která ho zároveň dělí na jednotlivé logické části, je sekce. Možné typy sekcí jsou TEXT, kde je uložen spustitelný kód programu, DATA obsahující inicializovaná data, BSS s neinicializovanými daty, DEBUG s ladicími informace a INFO s pomocnými informacemi.

Každá sekce má své volitelné jméno, které nemusí být unikátní. Počet sekcí je omezen rozsahem celého 32bitového čísla.

Podle typu souboru, který byl zmíněn u hlavičky programu, je sekce umístěna v paměti. Pokud se jedná o soubor bez relativních informací, musíme u každé sekce zadat její adresu v paměti, kam ji linker umístí. Naopak v případě souboru s relativními informacemi si linker určí sám, kam sekci umístí. Sekce se však nesmějí překrývat, jinak řečeno každý byte souboru může náležet maximálně jedné sekci.

V sekci se mohou nacházet i údaje o relokaci a řádkové informace, na které se podíváme podrobněji.

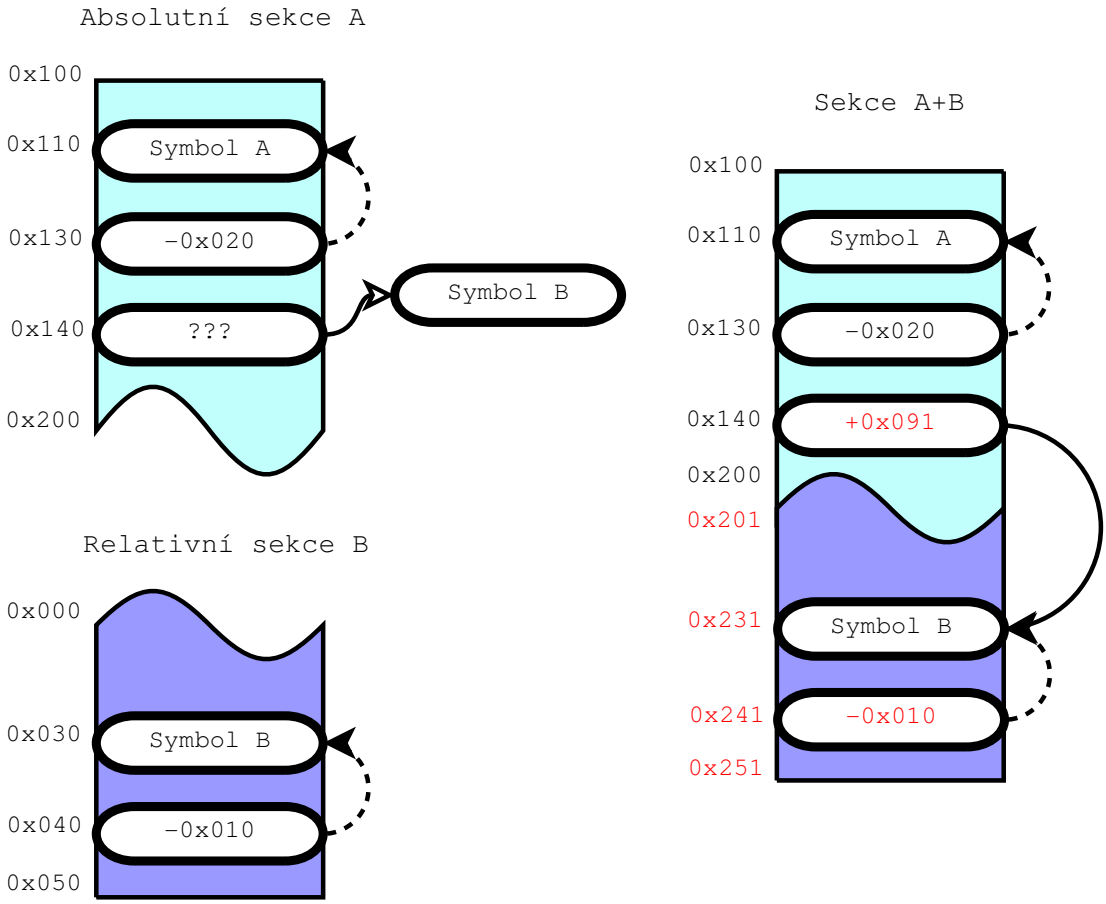
5.4 Relokace

Pro určení přesného místa k relokaci symbolu jsou použity následující údaje. Nejdříve musíme znát adresu relokovaného symbolu. Dále je to počet slov, který tuto adresu tvoří, a pokud se jedná o jedno slovo, můžeme ji dále upřesnit nejnižším a nejvyšším bitem tohoto slova. Díky tomu slze zajistit podporu i pro procesory, které uchovávají adresu některých symbolů přímo v operačním kódu.

Existují dva základní typy relokací, relativní a absolutní.

5.4.1 Relativní relokace

Odkazovaná adresa se udává relativně v podobně celého znaménkového čísla vzhledem k aktuální poloze. To má mj. výhodu v tom, že při přesunu sekce na jiné místo v paměti se nemusejí tyto relokace znovu přepočítávat. Spojení dvou sekcí s relativními relokacemi je uveden na obrázku 5.3.



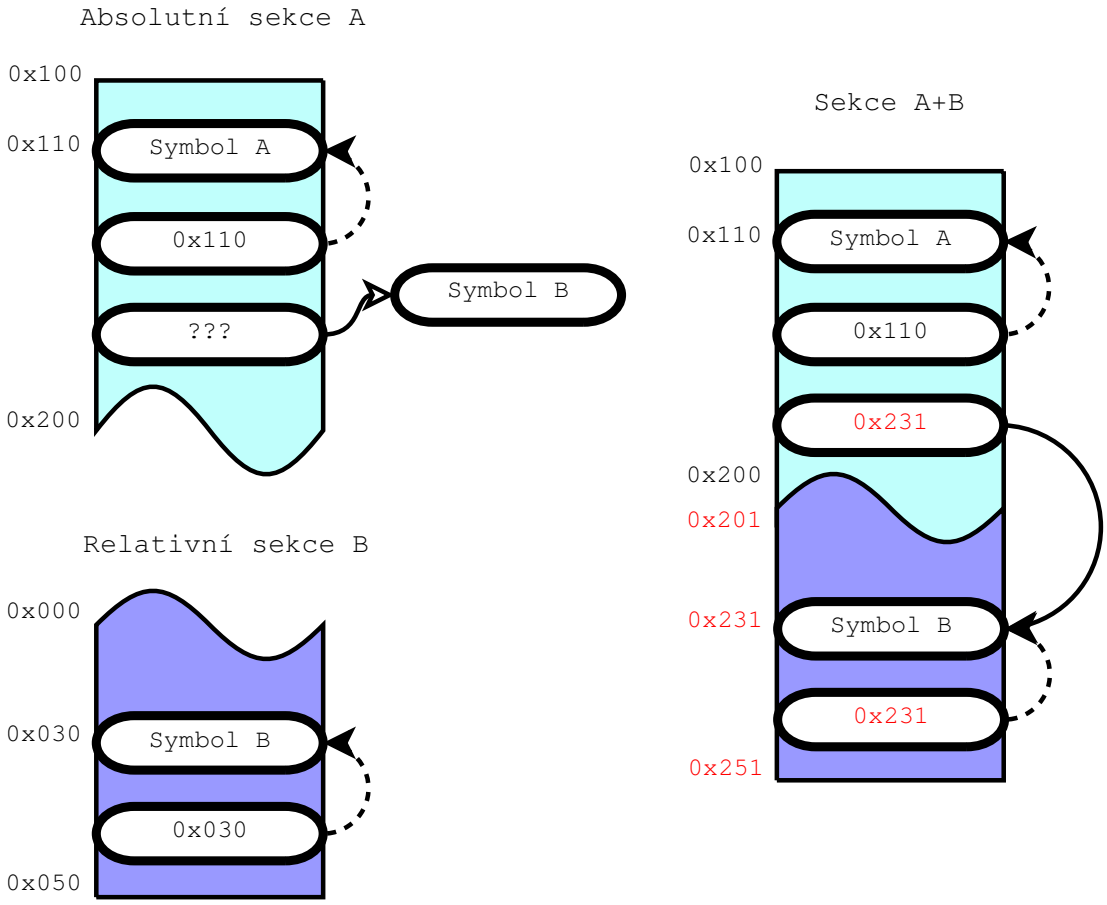
Obrázek 5.3: Příklad relativní relokace

5.4.2 Absolutní relokace

Sekce obsahují pouze odkazy s absolutní adresou symbolu v rámci své sekce ve tvaru celého nezáporného čísla. Při spojení absolutní a relativní sekce se adresy relativní sekce upraví tak, aby navazovaly na adresy sekce absolutní, a vypočítá se absolutní adresa symbolu, jak je zobrazeno na obrázku 5.4.

5.5 Řádkové informace

Slouží výhradně pro usnadnění ladění programu, na samotný běh programu nemají vliv. Pokud chceme mít možnost umístit na určitá místa v programu zarážky, program krokovat a zároveň mít zobrazen obsah proměnných, je nutné určitým způsobem propojit číslo řádku s adresou první instrukce tohoto řádku v binárním kódu. Jedna řádková informace obsahuje číslo řádku, odpovídající adresu instrukce na tomto řádku, jméno souboru a index symbolu, pokud se jedná například o návěští



Obrázek 5.4: Příklad absolutní relokace

nebo funkci. Číslovat se začíná od jedné, nultý řádek označuje výskyt symbolu. Index 0 tedy obsahuje index do tabulky symbolů.

Kapitola 6

Konverze mezi formáty CCOFF a ELF

V této kapitole bude popsán návrh a implementace konvertoru z formátu CCOFF do formátu ELF a zpět. Na obrázku 6.1 je znázoněn způsob využití GNU linkeru.

6.1 Existující konvertor

Peter Matula napsal v roce 2006 obousměrný konvertor CCOFF–BFD (Binary Format Description). Ten je však velmi omezený, byl vytvořen pouze pro architektury MIPS a ARM a v jeho současné podobě do něj nebylo možné připsat podporu relokací a 16bitových architektur. Díky použití knihovny LLVM-MC nebylo možné existující kódy konvertorů použít. Bylo zapotřebí vytvořit nové konvertory, ale v částech se daly ty existující využít pro inspiraci.

6.2 Soubor tříd ObjectFile

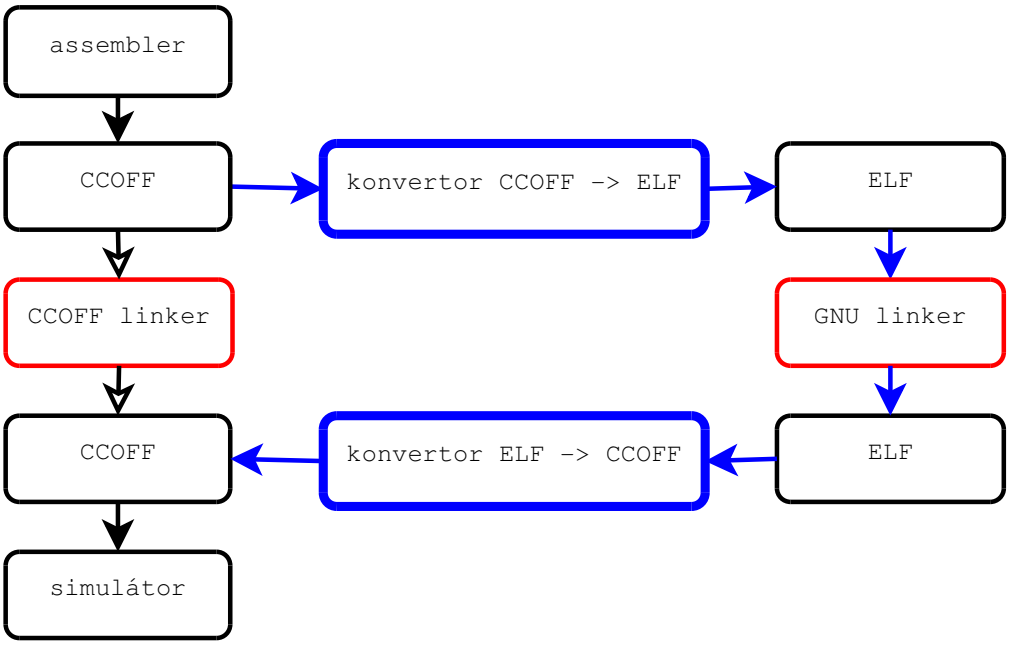
Pro vytvoření, načtení a rozparsování objektového souboru ve formátu CCOFF byla využita kolekce tříd ObjectFile modulu *dev*, kterou vytvořil v rámci své bakalářské práce Libor Vašíček [7]. Ve stručnosti popíši hlavní použité třídy.

`class CObjectFile`	definuje obecné vlastnosti objektového souboru, obsahuje odkazy na všechny sekce a tabulky symbolů,
`class CSection`	obsahuje informace o sekci a jejích datech, umožnuje mj. vytvářet nové sekce a přidávat, resp. odstraňovat data, relokace, symboly a řádkové informace,
`class CSymbol`	obsahuje jeden záznam tabulky symbolů,
`class CReference`	obsahuje jednu řádkovou informaci,
`class CRelocation`	specifikuje údaje a zacházení s relokacemi a
`class CSectionData`	manipuluje a ukládá data sekcí.

6.3 Návrh

Konvertor musí splňovat následující vlastnosti.

- Objektový návrh a naprogramování v jazyce C++ zajištující snadné budoucí úpravy a kompatibilitu se zbytkem projektu.



Obrázek 6.1: Využití GNU linkeru

- Výstupem konverze CCOFF−>ELF je validní ELF objektový soubor, který akceptují nástroje pro zobrazení jeho obsahu jako např. *readelf* nebo *objdump*.
- Při zpětné konverzi ELF−>CCOFF vznikne původní objektový soubor se stejným obsahem. Výjimku tvoří časová značka, která se vytvoří nově v době konverze, a konce řádků, které se mohou lišit při konverzi v prostředí jiného operačního systému.
- Prozatím stačí podpora pro 16bitovou a 32bitovou architekturu s little a big endianitou.
- Konverze proběhne v reálném čase.

Dále musí obsahovat funkce, které vytvoří jednotlivé části obj. souboru a naplní je správnými daty. Tyto části jsou

- hlavička souboru, která se nachází na začátku souboru na offsetu 0x0,
- hlavičky sekcí umístěné za samotnými sekcemi,
- , kromě sekcí konvertovaných přímo se musí vytvořit i sekce s tabulkou symbolů, textovými řetězci, nultá sekce, sekce ukládající názvy ostatních sekcí ve formě textových řetězců a sekce uchovávající řádkové informace,
- dále upravené relokační informace uložené v samostatné sekci,
- symboly,
- data sekcí,

Všechny tyto informace musí být zapsány do výstupního souboru ve správném pořadí a tvaru.

V následující části bude popsán postup při řešení a psaní konvertoru a dosažené výsledky. Kompletní popis dosazovaných hodnot lze nalézt v [5] a [6].

Atribut	Obsah
e_ident[EI_MAG0–3]	’0x7f’ ’E’ ’L’ ’F’
e_ident[EI_CLASS]	ELFCLASS32
e_ident[EI_DATA]	CObjectFile::LITTLE_ENDIAN => ELFDATA2LSB CObjectFile::BIG_ENDIAN => ELFDATA2MSB
e_ident[EI_VERSION]	EV_CURRENT
e_ident[EI_OSABI]	ELFOSABI_STANDALONE
e_ident[EI_ABIVERSION]	0
e_type	CObjectFile::X => ET_EXEC else => ET_REL
e_machine	32bit && LITTLE_ENDIAN => ARCH_32BIT_LITTLE 32bit && BIG_ENDIAN => ARCH_32BIT_BIG 16bit && LITTLE_ENDIAN => ARCH_16bit_LITTLE 16bit && BIG_ENDIAN => ARCH_16BIT_BIG
e_version	EV_CURRENT
e_entry	CObjectFile::GetEntryPoint()
e_phoff	0
e_shoff	sizeof (Elf32_Ehdr)
e_flags	CObjectFile::GetBytesPerWord()
e_ehsize	sizeof (Elf32_Ehdr)
e_phentsize	sizeof (Elf32_Phdr)
e_phnum	0
e_shentsize	sizeof (Elf32_Shdr)
e_shnum	SectionHeaders.size()
e_shstrndx	GetSectionIndexOfName(”.shstrtab”)

Tabulka 6.1: Konverze hlavičky souboru

6.4 Konverze hlavičky souboru

Nejméně komplikovanou částí je konverze hlavičky. V tabulce 6.1 je uvedeno, jakými daty naplnit položky hlavičky.

Položka `e_machine` standardně obsahuje jednu z položek výčtového typu, který obsahuje seznam velkého množství různých architektur. To ale není pro konvertor použitelné, protože je potřeba přesněji specifikovat počet bitů ve slově a počet bytů ve slově, který může být teoreticky libovolný. Proto se na hodnoty 22–25, které jsou nevyužité (resp. rezervovány pro budoucí architektury), využily pro specifikaci architektury. Prozatím stačí podpora pro 16 a 32 bitů, později ale nebude problém tuto nabídku rozšířit.

`e_flags` se využívá pro procesorově specifické příznaky. Takovou informaci ale není třeba ukládat, a proto je tato položka využita pro uložení počtu bytů ve slově.

6.5 Konverze sekce

Tabulka 6.2 ukazuje obsah hlavičky sekce.

Hlavičky sekcí jsou uchovány v datové struktuře `std::vector<Elf32_Shdr*>`. Dříve, než se začne se samotnou konverzí, musejí být vytvořeny následující sekce.

- **null section** se vytváří jako první a všechny její atributy mají hodnotu nula, dále

- **.shstrtab** uchovávající jména sekcí v textové podobě,
- **.strtab** uchovávající všechny textové řetězce v programu a nakonec
- **.symtab** ukládající informace o symbolech.

Pokud konvertovaná sekce obsahuje relokační informace, musí se vytvořit sekce se stejným názvem a předponou .rela pro každou takovou sekci. Relokace jsou vyřešeny trochu komplikovaněji, a proto se na ně zaměříme podrobněji.

6.6 Relokace

Jak jsem uvedl v kapitole 4 věnované ELF formátu, relokace obsahuje tři položky.

- **r_offset** označuje místo, kde se má provést relokace,
- **r_info** obsahuje informaci o indexu symbolu a typu relokace a
- **r_addend** obsahující konstantu pro upřesnění relokace.

Všechny typy relokací, které ELF podporuje, jsou architektonicky závislé a nedají se pro projekt využít. Navržené řešení je následující.

- **r_offset** bude stále označovat místo, kde se má provést relokace. Získáme ho pomocí funkce CRelocation::GetMemoryByteAddress().
- **r_info** bude v horních 24 bitech obsahovat index do tabulky symbolů a zbylých 8 bitů bude označovat typ relokace. Zatím jediný typ podporované relokace je `R_CODASIP_GENERAL` s hodnotou 0x7f.
- **r_addend** obsahuje index do sekce `.codasip_rel_info`, do které se budou ukládat struktury `Elf32_Codasip_Rel_Info`.

```
typedef struct {
  Elf32_Word r_type;          /* = CRelocation::GetType() */
  Elf32_Word r_word_count;    /* = CRelocation::GetByteAddressSize() */
  Elf32_Word r_first_bit_address_index;
                              /* = GetByteAddressFirstBitIndex() */
  Elf32_Word r_msb;           /* = CRelocation::GetHighBit() */
  Elf32_Word r_lsb;           /* = CRelocation::GetLowBit() */
  Elf32_Word r_addend;        /* = CRelocation::GetOffset() */
  Elf32_Word r_rshift;        /* = CRelocation::GetShiftAmount() */
  Elf32_Word r_mask;          /* = CRelocation::GetMask() */
} Elf32_Codasip_Rel_Info;
```

Díky tomuto řešení jsou uloženy všechny informace z formátu CCOFF. O jejich správnou interpretaci a zpracování se musí postarat upravený GNU linker.

6.7 Symboly

Informace o symbolu jsou uchovány ve struktuře `Elf32_Sym`. Nejdříve je do tabulky symbolů nahrán nultý symbol se všemi položkami vynulovanými a poté se může přejít k samotné konverzi z CCOFFu. Tabulka 6.3 ukazuje obsah této struktury.

Všechny symboly jsou uloženy v jedné společné tabulce symbolů. Uchovává je datová struktura `std::vector<Elf32_Sym* >`.

6.8 Data sekce

Data jsou v CCOFFu uložena v binární podobě, takže je stačí pouze překopírovat do příslušné sekce v ELFu. Binární data se uchovávají v datové struktuře `std::vector<char>`.

6.9 Řádkové informace

Tyto informace jsou uloženy v objektovém souboru ve formě řetězců oddělených znakem konce řádku. ELF obsahuje sekci `.line`, kam si obecně překladače ukládájí řádkové informace, ale kvůli eliminaci konfliktů v budoucnu je lepší vytvořit speciální sekci se zvoleným názvem `.codasip.line` a data do ní nakopírovat ve tvaru:

```
adresa\n
název souboru\n
číslo řádku\n
název symbolu (volitelný)\n
název sekce\n
```

6.10 Vytvoření validního objektového ELF souboru

Všechna získaná data, informace a datové struktury je nezbytné zapsat ve správném pořadí do výstupního souboru. Ukázky práce s ELF soubory jsou uvedeny např. v článcích [9] a [10].

Na offsetu 0x0 výstupního souboru začíná hlavička souboru. Za ní následují obsahy sekcí specifické pro ELF, tedy `.strtab`, `.shstrtab` a `.symtab`. Poté se ukládají obsahy sekcí konvertovaných z ELF souboru a za nimi sekce s relokacemi, `.codasip_rel_info` a `.codasip.line`. Pořadí je důležité z toho důvodu, aby při zpětné konverzi měly zpracované sekce k dispozici obsah těch předchozích a mohly se rovnou zpracovat i zpětné odkazy a indexy.

Po přilikování knihovny `libccofftoelf.a` stačí vytvořit objekt typu `CcoffToElf(string inputFile, string outputFile)` a konverze proběhne automaticky.

6.11 Zpětná konverze z formátu ELF do formátu CCOFF

Proces je velmi podobný tomu, jaký byl vysvětlen v předchozích sekcích, probíhá ale v opačném pořadí.

Nejdříve jsou načteny a zpracovány sekce `.strtab` a `.shstrtab`. První obsahuje jména symbolů a druhá jména sekcí. Následně je možné zpracovat sekci `.symtab` obsahující symboly. Díky tomu, že jsou už známy jména všech symbolů, je možné v objektovém souboru vytvořit tabulku symbolů a naplnit ji.

V další fázi se zpracují konvertované sekce. Je potřeba si dát pozor na to, aby byl správně nastaven typ sekce z parametrů `sh_type` a `sh_flags`. Následně se překopírují samotná data sekce. Jakmile jsou známa jména všech sekcí, nastaví se odkazy symbolů na správnou sekci.

V poslední fázi se zpracují relokace a řádkové informace. Pokud např. objektový soubor obsahoval sekci `.text` s relokacemi, všechny její relokace jsou v sekci `.rela.text`. Jak již bylo uvedeno v kapitole věnující se formátu ELF, relokace typu *rela*, která je využita, obsahuje atribut `r_addend` obsahující index do sekce `.codasip_rel_info`. Díky těmto dvěma sekcím a tabulce symbolů je možné zrekonstruovat původní relokace. Řádkové informace jsou jsou překonvertovány nakonec a svázány společně s příslušnou sekcí, ve které se původně vyskytovaly.

Atribut	Obsah
sh_name	SectionContents[".shstrtab"][<sectName>]
sh_type	CSection::BSS => SHT_NOBITS CSection::<else> => SHT_PROGBITS .shstrtab => SHT_STRTAB .strtab => SHT_STRTAB .symtab => SHT_SYMTAB .rel.<name> => SHT_RELA .codasip_rel_info => SHT_PROGBITS .codasip.line => SHT_PROGBITS
sh_flags	CSection::TEXT => SHF_ALLOC | SHF_EXECINSTR CSection::DATA => SHF_ALLOC | SHF_WRITE CSection::BSS => SHF_ALLOC | SHF_WRITE <else> => 0
sh_addr	CSection::GetMemoryAddress()
sh_offset	e_shoff + e_shnum ∗ e_shentsize
sh_size	SectionContent.size()
sh_link	GetSectionIndexOfName(".strtab")
sh_entsize	.symtab => sizeof(Elf32_Sym) .codasip_rel_info => sizeof(Elf32_Codasip_Rel_Info) else => 0

Tabulka 6.2: Konverze hlavičky sekce

Atribut	Obsah
st_name	SectionContents[StrTabContentsIndex]
st_value	CSymbol::GetMemoryAddress()
st_size	CSymbol::GetSymbolBlockSize()
st_type	CSymbol::FUNCTION => STT_FUNC CSymbol::OBJECT => STT_OBJECT CSymbol::FILE => STT_FILE
st_bind	CSymbol::PRIVATE => STB_LOCAL CSymbol::PUBLIC => STB_GLOBAL CSymbol::<else> => 0
st_info	ELF32_ST_INFO(st_bind, st_type)
st_other	0
st_shndx	CSymbol::PRIVATE => thisSectIndex CSymbol::LOCAL => thisSectIndex CSymbol::EXTERN => SHN_UNDEF CSymbol::ABSOLUTE => SHN_ABS CSymbol::COMMON => SHN_COMMON

Tabulka 6.3: Konverze symbolu

Kapitola 7

Závěr

Mým úkolem bylo navrhnout řešení podpory ladidích informací v sestavujícím programu a toto řešení implementovat. Po zvážení několika možností společně s vedoucím se zvolilo využití GNU linkeru. Společně s ním bylo nutné překonvertovat stávající formát CCOFF do formátu ELF.

Vytvořený konvertor dokáže převést soubor z formátu CCOFF do ELFu a nazpět tak, že se liší pouze časová značka a v případě použití jiného operačního systémů i konce řádků, což by ale nemělo činit při dalším zpracování problémy. Nástroje *readelf* a *objdump* dokáží zobrazit obsah všech částí ELF souboru, díky čemuž je zaručeno, že se jedná o validní soubor tohoto formátu.

Díky některým modifikacím formátu ELF, způsobu uložení dat a odlišnému formátu relokací a řádkových informací, je nutné GNU linker upravit tak, aby tyto změny akceptoval a dokázal s nimi správně pracovat. To je úkol, na kterém se aktuálně pracuje. Je pravděpodobné, že s těmito prováděnými úpravami bude nutné průběžně upravovat a rozšířovat i samotný konvertor. Tuto technickou zprávu a samotnou implementaci jsem pojal tak, aby nebyl problém změny v konvertoru provést. V budoucnu je plánováno, že se od formátu CCOFF upustí úplně a z assembleru se bude generovat přímo ELF.

Literatura

[1] LEVINE, J. R. *Linkers and Loaders*. San Diego, USA: Morgan Kaufmann, 2000. ISBN 978-1-55860-496-4.

[2] *System V Application Binary Interface*. : California, AT&T, 1997. Dostupné na: <http://www.sco.com/developers/devspecs/gabi41.pdf>.

[3] EAGER, J. *Introduction to the DWARF Debugging Format*. 2012 [cit. 1. května 2012]. Dostupné na: <http://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf>.

[4] *DWARF Debugging Information Format Version 4*. : DWARF Standards Committee, 2010-06-10. Dostupné na: <http://dwarfstd.org/doc/DWARF4.pdf>.

[5] WERVEN, J. R. *Linux Manual Reference Pages - ELF*. 1999.

[6] *Executable and Linkable Format (ELF)*. : TIS - Tool Interface Standards, 2005 [cit. 1. května 2012]. Dostupné na: <http://www.skyfree.org/linux/references/ELF_Format.pdf>.

[7] VAŠÍČEK, L. *Návrh struktury obecného assembleru a zpětného assembleru*. : FIT VUT v Brně, 2006.

[8] KOLÁŘ, D. *Návrh vstupního formátu pro assembler a linker*. : FIT VUT v Brně, 2004.

[9] YOUNGDALE, E. The ELF Object File Format: Introduction [online]. *Linux Journal*. 1995-04-01 [cit. 1. května 2012]. Dostupné na: <http://www.linuxjournal.com/article/1059>.

[10] YOUNGDALE, E. The ELF Object File Format by Dissection [online]. *Linux Journal*. 1995-05-01 [cit. 1. května 2012]. Dostupné na: <http://www.linuxjournal.com/article/1060>.